Distributed Programming using Role-Parametric Session Types in Go

David Castro[1], Raymond Hu[1], Sung-Shik Jongmans[1,2], Nicholas Ng[1], Nobuko Yoshida[1]

[1] Imperial College London
[2] Open University of the Netherlands

Distributed Programming using Role-Parametric Session Types in Go

David Castro[1], Raymond Hu[1], Sung-Shik Jongmans[1,2],
Nicholas Ng[1], Nobuko Yoshida[1]

[1] Imperial College London
[2] Open University of the Netherlands

Distributed Programming using Role-Parametric Session Types in Go

David Castro[1], Raymond Hu[1], Sung-Shik Jongmans[1,2], Nicholas Ng[1], Nobuko Yoshida[1]

[1] Imperial College London

[2] Open University of the Netherlands

Introduction (distributed programming in Go)

Long-term research agenda:

Development of theory and tools
to help programmers write
safe concurrent programs

Introduction (distributed programming in Go)

Long-term research agenda:

> Development of theory and tools
> to help Go programmers write
> **safe** concurrent Go programs

[CC'16, POPL'17, ICSE'18]

Introduction (distributed programming in Go)

- (a) Modern, popular systems language

Introduction (distributed programming in Go)

- (a) Modern, popular systems language
- (b) Primacy of CSP-based concurrency features
  - Lightweight threads, called *goroutines*
  - Higher-order, typed native channels (across shared memory)
  - First-order, untyped API channels (across a network)

Introduction (distributed programming in Go)

- (a) Modern, popular systems language
- (b) Primacy of CSP-based concurrency features
  - Lightweight threads, called *goroutines*
  - Higher-order, typed native channels (across shared memory)
  - First-order, untyped API channels (across a network)
- (c) Survey: "Users least agreed that they are able to effectively debug uses of Go's concurrency features"

Introduction (distributed programming in Go)

multiparty session types? [POPL'08]

- (a) Modern, popular systems language
- (b) Primacy of CSP-based concurrency features
  - Lightweight threads, called *goroutines*
  - Higher-order, typed native channels (across shared memory)
  - First-order, untyped API channels (across a network)
- (c) Survey: "Users least agreed that they are able to effectively debug uses of Go's concurrency features"

Introduction (distributed programming in Go)

Motivating example: `htcat`

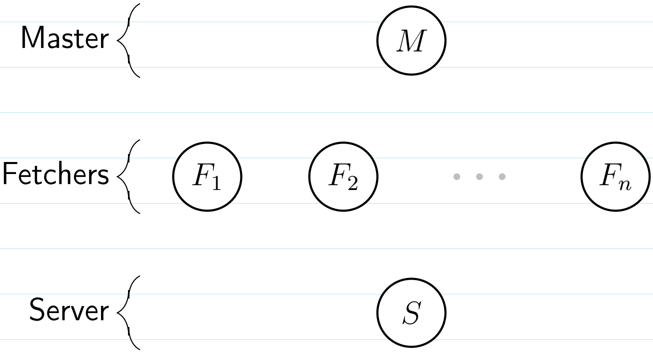
(`https://github.com/htcat/htcat`)

Parallel downloader of webpages

Post-factum verification very difficult

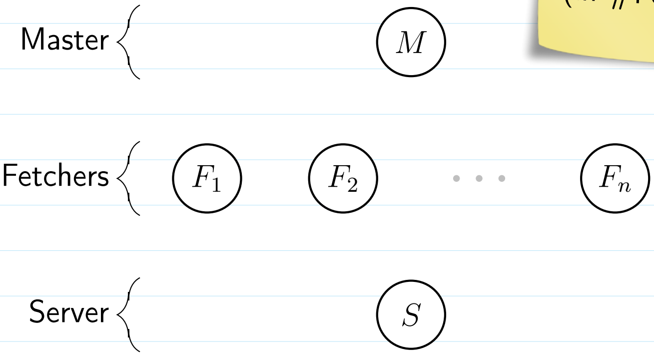
Our *safe-by-construction* version: PGet ( )

Introduction (distributed programming in Go)



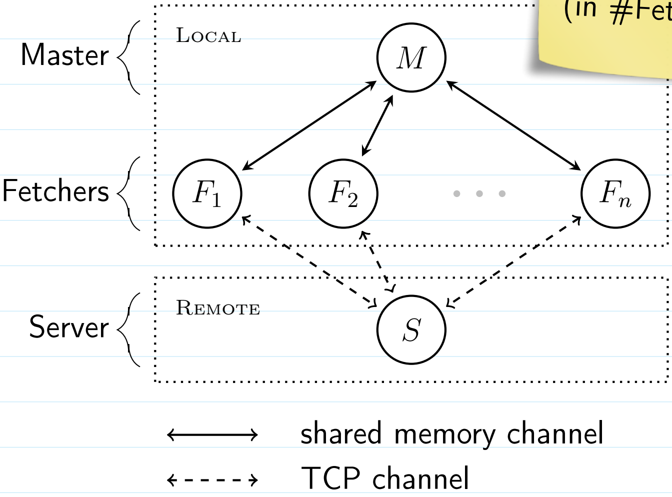
Introduction (distributed programming in Go)

feature 1: parameterisation (in #Fetchers)



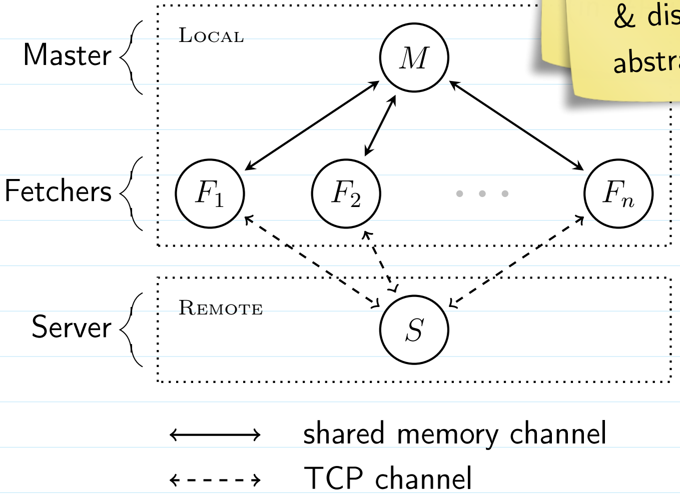
Introduction (distributed programming in Go)

feature 1:
parameterisation
(in #Fetchers)

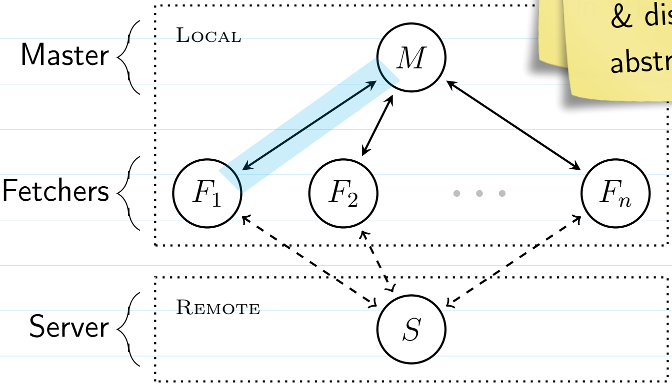


Introduction (distributed programming in Go)

feature 2:
mixed transports
& disparate
abstractions



Introduction (distributed programming in Go)

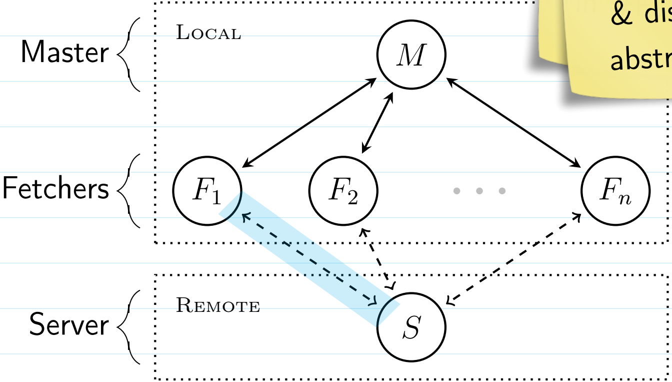


feature 2:
mixed transports
& disparate
abstractions

$\mathtt{M} \longrightarrow \mathtt{F}_1$ `:GetSize(string)`

Introduction (distributed programming in Go)

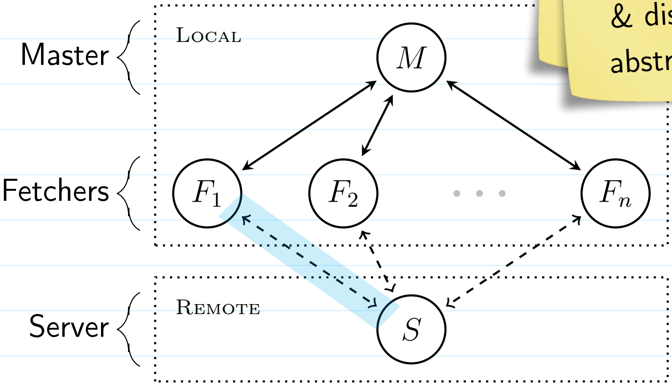
feature 2:
mixed transports
& disparate
abstractions



$F_1 \longrightarrow S$: `HttpReq(byte[])`

Introduction (distributed programming in Go)

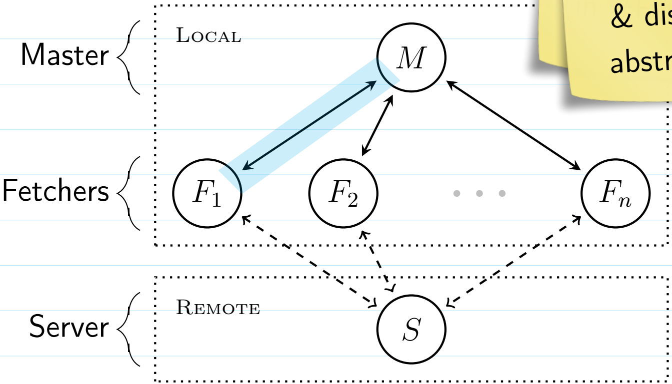
feature 2:
mixed transports
& disparate
abstractions



$$F_1 \rightarrow S : \texttt{HttpReq(byte[])} \,.\, S \rightarrow F_1 : \texttt{HttpRes(byte[])}$$

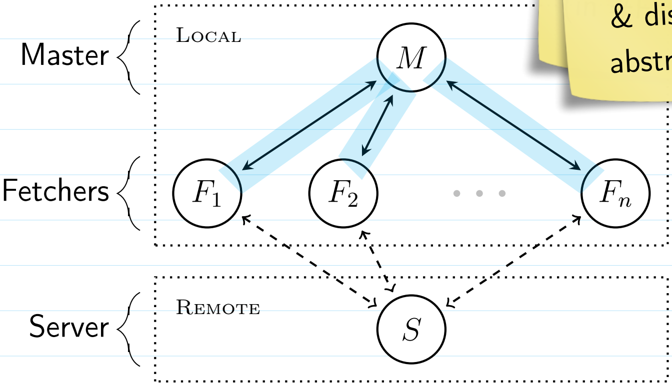
Introduction (distributed programming in Go)

feature 2:
mixed transports
& disparate
abstractions



$F_1 \longrightarrow M$: Size(int)

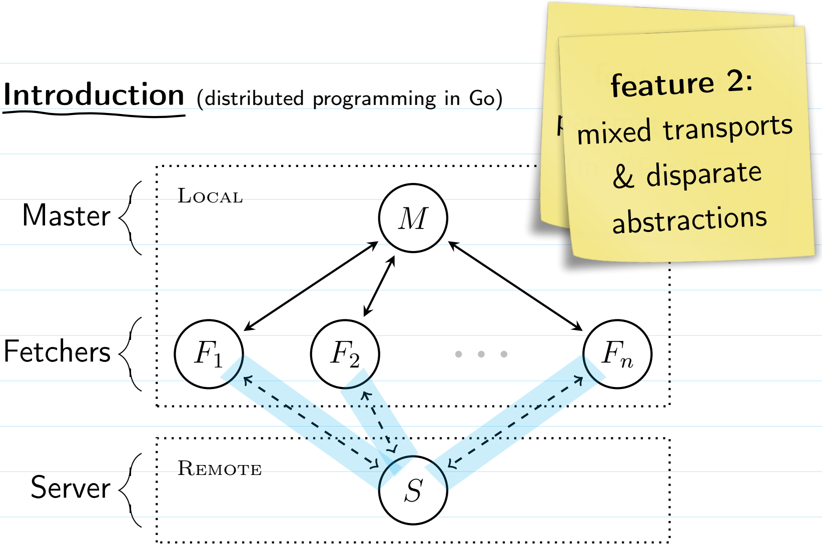
Introduction (distributed programming in Go)



feature 2: mixed transports & disparate abstractions

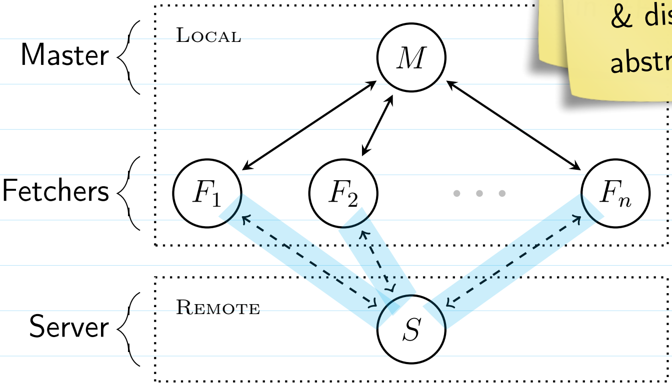
M ⟶ F[1..n] :GetData(string,int,int)

Introduction (distributed programming in Go)



`F[1..n] → S : HttpReq(byte[])`

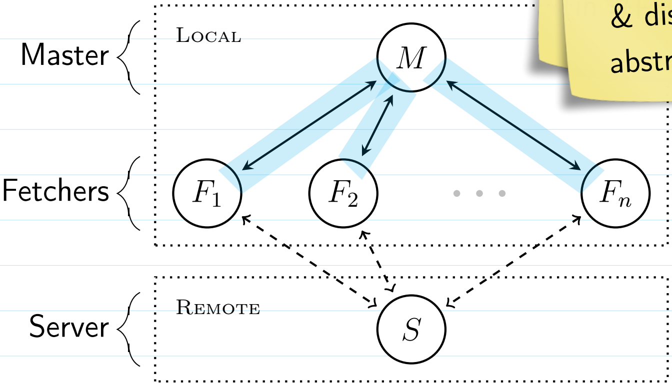
Introduction (distributed programming in Go)



feature 2:
mixed transports
& disparate
abstractions

$\texttt{F[1..n]} \longrightarrow \texttt{S:HttpReq(byte[])}$. $\texttt{S} \longrightarrow \texttt{F[1..n]:HttpRes(byte[])}$

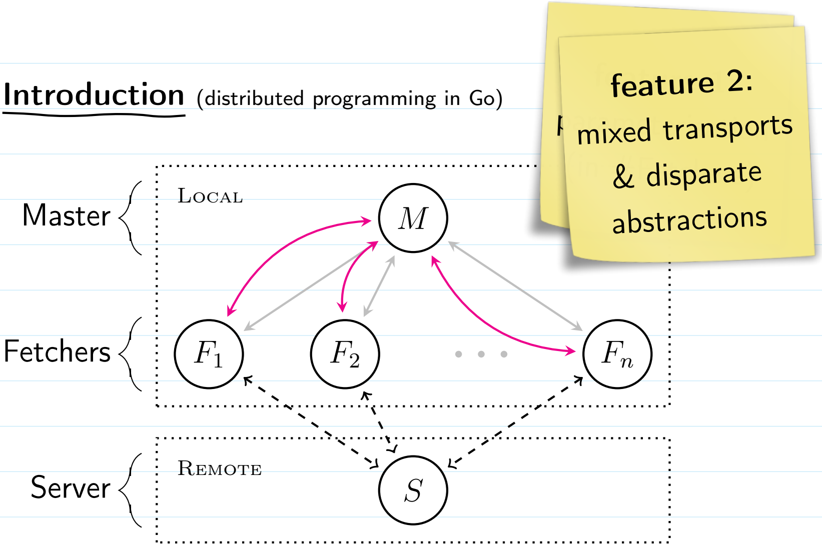
Introduction (distributed programming in Go)



feature 2: mixed transports & disparate abstractions

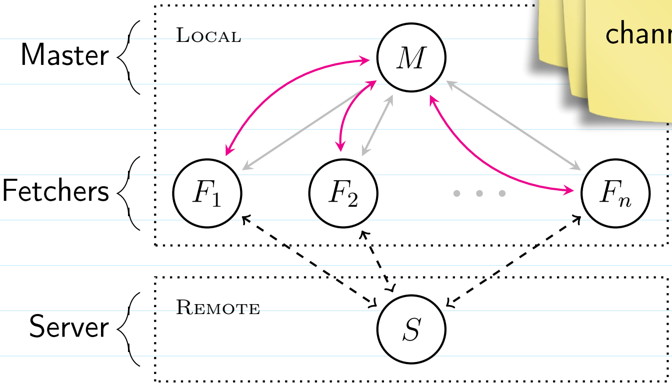
`F[1..n] → M : Data(string, chan)`

Introduction (distributed programming in Go)



F[1..n] ⟶ M :Data(string, **chan**)

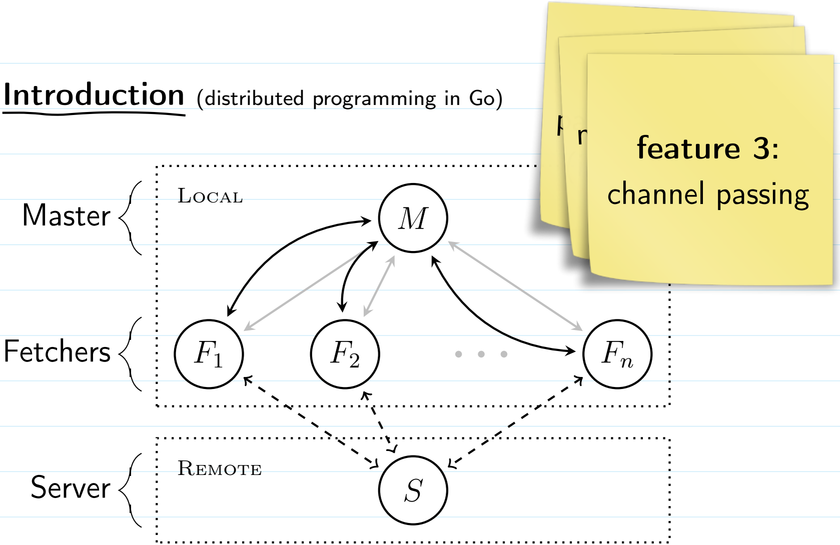
Introduction (distributed programming in Go)



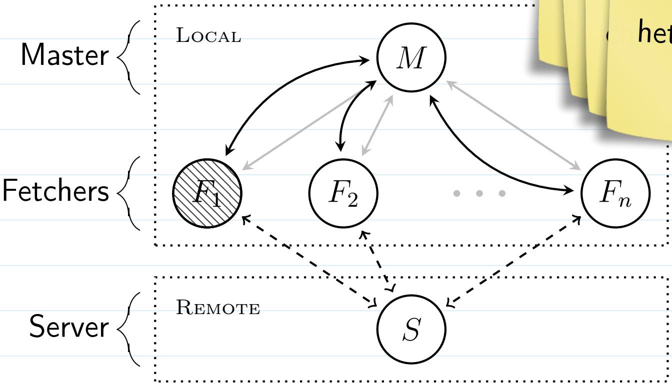
`F[1..n] → M:Data(string,chan)`

feature 3:
channel passing

Introduction (distributed programming in Go)



Introduction (distributed programming in Go)



feature 4:
heterogeneous
roles

Introduction (distributed programming in Go)

Features:

- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles

Introduction (distributed programming in Go)

Features:

- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles

Challenges (*safety*):

- Protocol compliance
- Deadlock-freedom

Introduction (distributed programming in Go)

Features:

- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles

Challenges (*safety*):

- Protocol compliance ✓
- Deadlock-freedom ✓

multiparty session types:

Introduction (distributed programming in Go)

Features:

- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles

Challenges (*safety*):

- Protocol compliance ✓
- Deadlock-freedom ✓

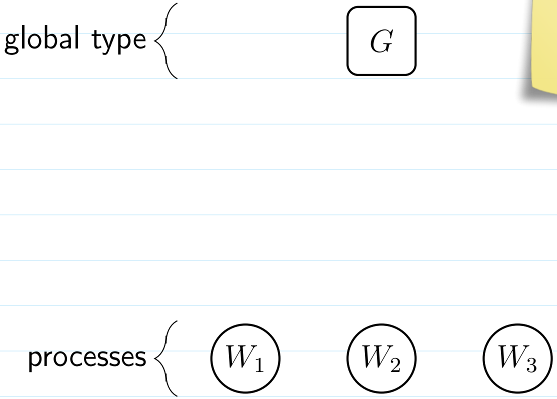
multiparty session types:

real programs need more expressive theory and impl.

Introduction (multiparty session types; MPST)

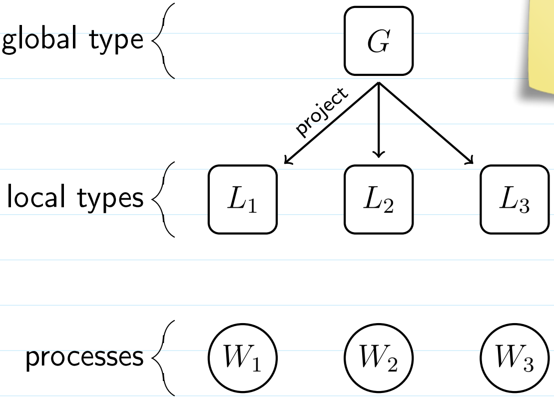
processes

Introduction (multiparty session types; MPST)



$$G = \texttt{W}_1 \to \texttt{W}_2 : \texttt{Int} \,.\, \texttt{W}_2 \to \texttt{W}_3 : \texttt{Bool}$$

Introduction (multiparty session types; MPST)



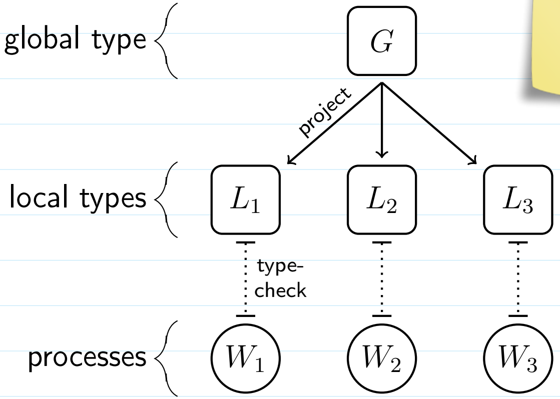
$$G = \texttt{W}_1 \to \texttt{W}_2 : \texttt{Int} \,.\, \texttt{W}_2 \to \texttt{W}_3 : \texttt{Bool}$$

$$L_1 = \texttt{W}_2 ! \texttt{Int}$$

$$L_2 = \texttt{W}_1 ? \texttt{Int} \,.\, \texttt{W}_3 ! \texttt{Bool}$$

$$L_3 = \texttt{W}_2 ? \texttt{Bool}$$

Introduction (multiparty session types; MPST)



$$G = \mathtt{W}_1 \to \mathtt{W}_2 : \mathtt{Int}\,.\ \mathtt{W}_2 \to \mathtt{W}_3 : \mathtt{Bool}$$

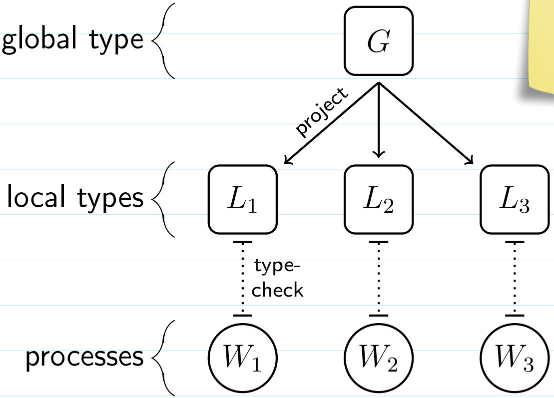
$$L_1 = \mathtt{W}_2 ! \mathtt{Int}$$

$$L_2 = \mathtt{W}_1 ? \mathtt{Int}\,.\ \mathtt{W}_3 ! \mathtt{Bool}$$

$$L_3 = \mathtt{W}_2 ? \mathtt{Bool}$$

Introduction (multiparty session types; MPST)

$$G = \mathtt{W}_1 \to \mathtt{W}_2 : \mathtt{Int}\ .\ \mathtt{W}_2 \to \mathtt{W}_3 : \mathtt{Bool}$$



$$L_1 = \mathtt{W}_2\mathtt{!Int}$$

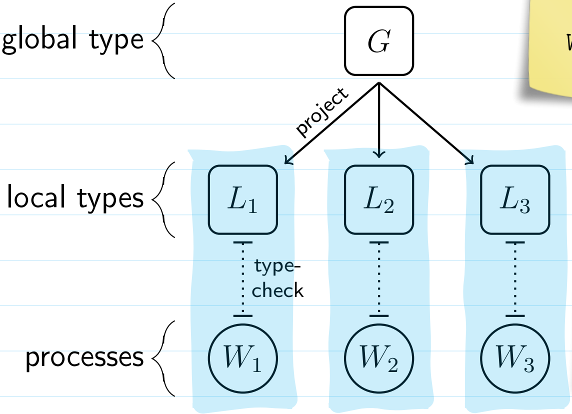
$$L_2 = \mathtt{W}_1\mathtt{?Int}\ .\ \mathtt{W}_3\mathtt{!Bool}$$

$$L_3 = \mathtt{W}_2\mathtt{?Bool}$$

well-typed $\Rightarrow$ protocol compliance $\wedge$ deadlock-freedom

Introduction (multiparty session types; MPST)

$$G = \texttt{W}_1 \to \texttt{W}_2 : \texttt{Int}\,.\; \texttt{W}_2 \to \texttt{W}_3 : \texttt{Bool}$$



$$L_1 = \texttt{W}_2\,!\,\texttt{Int}$$
$$L_2 = \texttt{W}_1\,?\,\texttt{Int}\,.\; \texttt{W}_3\,!\,\texttt{Bool}$$
$$L_3 = \texttt{W}_2\,?\,\texttt{Bool}$$

well-typed $\Rightarrow$ protocol compliance $\wedge$ deadlock-freedom

Contributions

Theory:

- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness

Contributions

Theory:

- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness

Implementation:

- Extension to **Scribble** [FASE'16, FASE'17]
- Artifact (reusable and available)

Contributions

Theory:

- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness

Implementation:

- Extension to **Scribble** [FASE'16, FASE'17]
- Artifact (reusable and available)

Evaluation:

- Competitive performance
- Wide applicability

Theory

Easy part:
Parameterisation

$$G = \texttt{\textbf{foreach}}\ \texttt{W}[\texttt{i:1..n-1}, \texttt{j:2..n}]\ \texttt{\textbf{do}}\ \texttt{W}[\texttt{i}] \rightarrow \texttt{W}[\texttt{j}]\ \texttt{\textbf{:}Msg}$$

Theory

Easy part:
Parameterisation

$G =$ **foreach** `W[i:1..n-1, j:2..n]` **do** `W[i]` $\to$ `W[j]` **:**`Msg`

Hard part:
Role heterogeneity

How to infer from G there exist three *role variants*?
(first Worker; middle Workers; last Worker)

Theory

$$G = \textbf{foreach}\ \texttt{W}[\texttt{i:1..n-1}, \texttt{j:2..n}]\ \textbf{do}\ \texttt{W}[\texttt{i}] \to \texttt{W}[\texttt{j}]\ \texttt{:Msg}$$

Key insight: Behaviour of Worker x is determined by the *intervals* in which x occurs (i.e., if x and y are contained in the same intervals, Workers x and y behave the same)

Theory

$$G = \texttt{\textbf{foreach}}\ \texttt{W}[\texttt{i:1..n-1}, \texttt{j:2..n}]\ \texttt{\textbf{do}}\ \texttt{W}[\texttt{i}] \rightarrow \texttt{W}[\texttt{j}]\ \texttt{:Msg}$$

Key insight: Behaviour of Worker x is determined by the *intervals* in which x occurs (i.e., if x and y are contained in the same intervals, Workers x and y behave the same)

$$x \in \texttt{1..n-1} \wedge x \in \texttt{2..n} \Rightarrow x \in \texttt{2..n-1} \qquad \text{(middle Worker)}$$
$$x \in \texttt{1..n-1} \wedge x \notin \texttt{2..n} \Rightarrow x = \texttt{1} \qquad \text{(first Worker)}$$
$$x \notin \texttt{1..n-1} \wedge x \in \texttt{2..n} \Rightarrow x = \texttt{n} \qquad \text{(last Worker)}$$
$$x \notin \texttt{1..n-1} \wedge x \notin \texttt{2..n} \Rightarrow \bot$$

Theory

- 1. Infer role variants as triples $r[D, \bar{D}]$, where:
  - r is a *role name*
  - D is a set of *intervals*
  - $\bar{D}$ is a set of *"co-intervals"*

Theory

- 1. Infer role variants as triples $r[D, \bar{D}]$, where:
  - r is a *role name*
  - D is a set of *intervals*
  - $\bar{D}$ is a set of *"co-intervals"*

- 2. Project G onto inferred role variants, e.g.:

$$G \restriction \mathtt{W}[\{\mathtt{1..n\text{-}1}, \mathtt{2..n}\}, \emptyset] = \mathtt{W}[\mathbf{self}\text{-}1]\mathbf{?}\mathtt{Msg}\ \mathbf{.}\ \mathtt{W}[\mathbf{self}\text{+}1]\mathbf{!}\mathtt{Msg}$$

$$G \restriction \mathtt{W}[\{\mathtt{1..n\text{-}1}\}, \{\mathtt{2..n}\}] = \mathtt{W}[\mathbf{self}\text{+}1]\mathbf{!}\mathtt{Msg}$$

$$G \restriction \mathtt{W}[\{\mathtt{2..n}\}, \{\mathtt{1..n\text{-}1}\}] = \mathtt{W}[\mathbf{self}\text{-}1]\mathbf{?}\mathtt{Msg}$$

Theory

Theorem: Inferring role variants is decidable

Theorem: Checking well-formedness is decidable

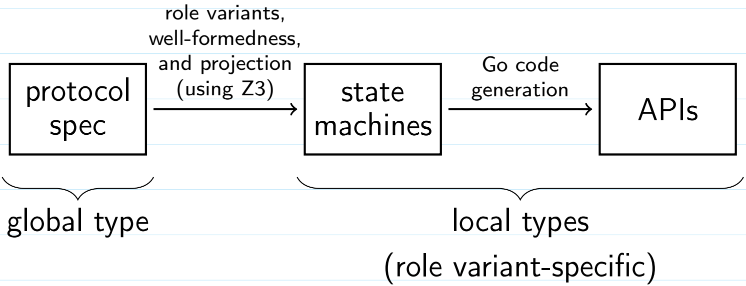
Theorem: Projecting well-formed global types is semantics-preserving, i.e., correct

$$\begin{array}{ccccc}
\mathbb{G} & \xrightarrow{\langle\!\langle \cdot \rangle\!\rangle} & \mathbb{G}_{\text{restr}} & \xrightarrow{[\![\cdot]\!]} & \mathbb{G}_{\text{orig}} \\
\downarrow \upharpoonright & \circlearrowleft & \downarrow \upharpoonright & \circlearrowleft & \downarrow \upharpoonright_{\text{orig}} \\
\mathbb{L} & \xrightarrow{\langle\!\langle \cdot \rangle\!\rangle} & \mathbb{L}_{\text{restr}} & \xrightarrow{[\![\cdot]\!]} & \mathbb{L}_{\text{orig}}
\end{array}$$

Implementation

Extension of protocol description language **Scribble**

(`http://www.scribble.org`)

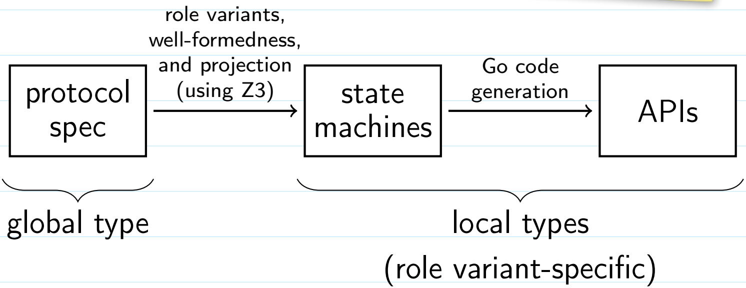


Implementation

Extension of protocol description l

(http://www.scribbl

APIs guide programmer towards safety



Implementation

(demo video)

Implementation

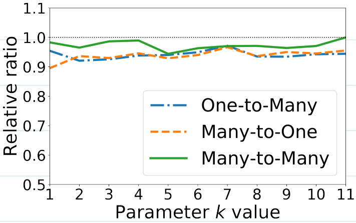
Guarantees:

- Protocol compliance
- Deadlock-freedom (up to “protocol-unrelated” program behaviour, premature termination, and delegation)

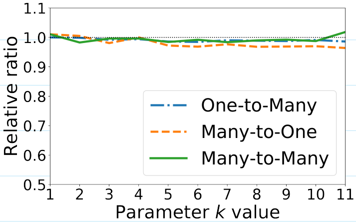
Achieved through:

- Native Go typing
- Lightweight run-time checks for linearity

Evaluation (benchmarks)



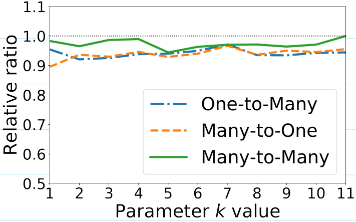
SHM



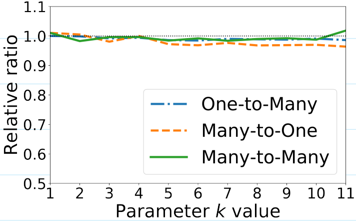
TCP

- Microbenchmarks
  - Speed-up (t_1/t_2) of **Scribble** (t_2) vs. native Go (t_1)
  - Per communication: $\sim$20ns

Evaluation (benchmarks)



SHM



TCP

- Microbenchmarks
  - Speed-up (t_1/t_2) of **Scribble** (t_2) vs. native Go (t_1)
  - Per communication: $\sim$20ns
- Computer Language Benchmark Games (CLBG)

Evaluation (expressiveness)

Core I/O patterns

	Pt	Sc	Ga	FE
1. One-to-Many (§ 6.1)		●		○
2. Many-to-One (§ 6.1)			●	○
3. Many-to-Many (§ 6.1)		●	●	○

Above, ○ are possible alt. implementations

		Pt	Sc	Ga	FE	Pipe	MS	PP	Rec	Del
Parallel Topologies	4. Pipeline (§ 4)	●				●				
	5. Ring (§ 3; 4)	●			●	●		●	●	
	6. Hadamard (§ 4)	●			●					
	7. Mesh (§ 4)	●			●	●				
	8. Fork-Join		●	●						
Applications	9. Pget[2] (□ is the difference between the two versions in § 3.2; § 3.3)	●	●	●	□					●
	10. Vickrey auction (Supplement, § IV.1.2)	●	●	●	●		●		●	
	11. Jacobi solution of discrete Poisson equation. [Bejleri et al. 2009]	●	●	●	●		●		●	
	12. *n*-body simulation (based on Ring) [Bejleri et al. 2009]	●			●	●		●	●	
	13. Iterative linear equation solver (based on Mesh) [Ng and Yoshida 2015]	●	●		●	●	●		●	
	14. `k-nucleotide` [Gouy 2017] (§ 6.1)		●	●						
	15. `regex-redux` [Gouy 2017] (§ 6.1)		●	●						
	16. `spectral-norm` [Gouy 2017] (§ 6.1)		●	●			●		●	
	17. Fibonacci [Lange et al. 2017]	●			●					
	18. Quote-Request [Austin et al. 2004; Ng and Yoshida 2015]		●	●	●		●		●	
	19. P2P multiplayer game [Scalas et al. 2017]	●			●	●		●	●	●
	20. Web Crawler [Akhmadeev 2016; Neykova and Yoshida 2017]	●	●	●	●					
	21. *n*-buyers [Coppo et al. 2016; Honda et al. 2016]	●			●		●		●	

Pt: point-to-point; Sc: Scatter; Ga: Gather; FE: Foreach; Pipe: Pipeline; MS: MS choices; PP: PP choices; Rec: Recursion; Del: Delegation

21 patterns, topologies, and applications

(each uses various features of our framework)

Conclusion

Also in the paper:

- Branching, selection, recursion, merge
- Implementation
  - Transport independence
  - Linearity checks (Go does not have linear types)

Technical report with all details:

https://www.doc.ic.ac.uk/research/technicalreports/2018/#4

Conclusion

Theory:

- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness

Implementation:

- Extension to **Scribble**
- Artifact (reusable and available)

Evaluation:

- Competitive performance
- Wide applicability